www.codeguru.com/Cpp/Cpp/cpp_mfc/pointers/article.php/c829/

Back to Article

Home >> **Visual C++ / C++** >> **C++** >> **C++ & MFC** >> **Pointers**

Smart Pointer (with Object Level Thread Synchronization '& Reference Counting Garbage Collection)

Rating: none

Stefan Tchekanov (view profile)
January 25, 2000

(continued)

What is new

- SmartPtr objects can accept different pointer types - details
- Passing a SmartPtr object as function parameter or function result is as efficient as passing an integer value or regular pointer - details.

Introduction

The SmartPtr class is a generic wrapper that encapsulates pointers. It keeps track of the total number of reference counts and performs garbage collection when the pointed object is no more referenced by any "pointer". In addition SmartPtr can do Object Level Thread Synchronization for pointed objects. This means that calling the members of the pointed object is enclosed in Windows critical section and only one thread at a time can use the object.

SmartPtr stores reference counter and critical section in separate memory block - not in the pointed object. This causes

that you don't have to inherit your classes from special base one to be able to use SmartPtr as a pointer. And one other think is that you could use SmartPtr to point objects of scalar types - int, short, long, double ....

SmartPtr is implemented as a template class. Because it has three template parameters it is annoying to specify all of them each time. This is the reason I have defined three additional template classes inherited from SmartPtr with appropriate default template parameters to get three diferent behaviours:

- **Reference Counting Garbage Collection** - SmartPtr will free dynamically allocated memory automatically.
- **Synchronized Access without Reference Counting Garbage Collection** - SmartPtr will perform only Thread synchronization. You should free dynamically allocated memory in your own.
- **Synchronized Access with Reference Counting Garbage Collection** - SmartPtr will perform thread synchronization and will free dynamically allocated memory for you.

These are the classes:

Class Name	Description
SmartPtrBase	The SmartPtrBase class is a non template base class for the SmartPtr class. It should never be used directly. SmartPtr uses it to perform appropriate type casting
SmartPtr	The SmartPtr class is a template base class for the next classes
RefCountPtr	Smart pointers that perform only Reference Counting Garbage Collection
SyncPtr	Smart pointers that perform only Synchronized Access without Reference Counting Garbage Collection
SyncRefCountPtr	Smart pointers that perform both Synchronized Access and Reference Counting Garbage Collection

All of these classes are equal in their implementation. The different behaviour is achieved via different default template parameters.You could use any of these four classes to get any of the three behaviours but then you should specify all parameters.

The following classes are used inside the SmartPtr implementation and SHOULD NEVER be used directly.

- CRefCountRep
- CSyncAccessRep
- CSyncRefCountRep
- CSyncAccess

Examples of how to use SmartPtr

You should include the file SmartPtr.h in your project.

I usually include this line in my StdAfx.h file.

```
#include "SmartPtr.h"
```

Let's have a class CSomething

1. Reference Counting Garbage Collection on CSomething class

```
class   CSomething {
        CSomething();
        ~CSomething();
.....
        void    do();
};
```

```
typedef RefCountPtr      LPSOMETHING;


void    TestFunct() {
        LPSOMETHING     p1 = new CSomething;
        LPSOMETHING     p2 = p1;

        if( p1 == NULL ) {
        ....
        }

        p2->do();
        p1 = NULL;

}//      Here the object pointed by p2 WILL BE destroyed automatically
////////////////////////////////////////////////////////////////////////////
```

2. Object Level Thread Synchronization for objects of CSomething

```
class   CSomething {
        CSomething();
        ~CSomething();
.....
        void    do();
};

typedef SyncPtr  LPSOMETHING;

void    TestFunct() {
        LPSOMETHING     p1 = new CSomething;
        LPSOMETHING     p2 = p1;

        if( p1.IsNull() ) {
        ....
        }
        StartThread( p1 );

        p2->do();       //      Synchronized with the other thread
        p1 = NULL;

}       //      Here the object pointed by p2 will NOT be destroyed automatically

void    ThreadFunc( LPSOMETHING p ) {
        p->do();        //      Synchronized with the other thread
}//      Here the object pointed by p will NOT be destroyed automatically
////////////////////////////////////////////////////////////////////////////
```

In this example you will get memory leaks, but the two threads will be synchronized when trying to call object's members. It is your care to free dynamically allocated memory.

3. Object Level Thread Synchronization and Reference Counting Garbage Collection for objects of CSomething

```
class   CSomething {
        CSomething();
        ~CSomething();
.....
```

```
        void    do();
};

typedef SyncRefCountPtr  LPSOMETHING;

void    TestFunct() {
        LPSOMETHING     p1 = new CSomething;
        LPSOMETHING     p2 = p1;

        if( p1.IsNull() ) {
        ....
        }
        StartThread( p1 );

        p2->do();       //      Synchronized with the other thread
        p1 = NULL;

}//     Here the object pointed by p2 WILL BE destroyed automatically
 //     if p in ThreadFunc has already released the object

void    ThreadFunc( LPSOMETHING p ) {
        p->do();        //      Synchronized with the other thread
}//     Here the object pointed by p WILL BE destroyed automatically
 //     if p2 in TestFunc has already released the object
////////////////////////////////////////////////////////////////////////////
```

In this example you will not get memory leaks and the two threads will be synchronized when trying to call object's members. You don't have to free dynamically allocated memory. SmartPtr will do it for you.

How does SmartPtr work?

The definition of SmartPtr is:

```
template, class ACCESS = T*>
class SmartPtr : public SmartPtrBase {
.....
};
```

Where *T* is the type of the pointed objects, *REP* is the representation class used to handle the pointers and *ACCESS* is the class used to get thread safe access to the underlying real pointer.

There is nothing interesting about Reference Counting Garbage Collection. It is a standard implementation. The only interesting thing is that the reference counter and the real pointer are stored in the representation object instead of the pointed object.

The more interesting thing is how Object Level Thread Synchronization works. Let's look at the definition of SyncPtr class:

```
template, class ACCESS=CSyncAccess  >
class SyncPtr {
.....
        ACCESS  operator -> ();
};
```

Look at the reference operator. It returns object of type ACCESS, which by default is of type CSyncAccess. The trick is that if the reference operator returns something different than a real pointer,

the compiler calls the operator ->() of the returned object. If it returns again something different than a real pointer compiler calls the returned object operator ->(). And this continues till some reference operator returns a real pointer like T*. Well, to get object level synchronization I use this behaviour of the reference operator. I have defined a class CSyncAccess:

```
template
class   CSyncAccess {
.............
        virtual ~CSyncAccess();
        T*      operator -> ();
};
////////////////////////////////////////////////////////////////////////////
```

The representation class used in this scenario has a member of type CRITICAL_SECTION.

```
template
class CSyncAccessRep {
.............
        CRITICAL_SECTION        m_CriticalSection;
};
////////////////////////////////////////////////////////////////////////////
```

And now let's look at the example code:

```
typedef SyncPtr  LPSOMETHING;
LPSOMETHING      p = new CSomething;

p->do();
```

What's going when p->do() is called?

1. The compiler calls SyncPtr::operator->() which returns object of type CSyncAccess. This object is temporary and is stored on top of the stack. In its constructor the critical section object is initialized.
2. The compiler calls CSyncAccess:operator->() which owns the critical section and thus protects other threads to own it and then returns the real pointer to the pointed object
3. The compiler calls the method do() of the pointed object
4. The compiler destroys CSyncAccess object which is on the stack. This calls its destructor where the critical section is released and other threads are free to own it.

The good thing about SmartPtr is that pointed objects do not have to be inherited from any special base class as in many other reference counting implementations. The reference counter and the real object pointer are stored in dinamically allocated objects of types CRefCountRep, CSyncAccessRep and CSyncRefCountRep (called representation objects) depending on what kind of SmartPtr we have. These objects are allocated and freed by SmartPtr. This approach allows us to have SmartPtr for objects of any classes, not only for inherited from special base class. Well, this is the weakness of the SmartPtr too. Imagine that you have a piece of code like this:

```
LPSOMETHING      p1 = new CSomething;
CSomething*      p2 = (CSomething*)p1;
LPSOMETHING      p3 = p2;
```

As result, at the end of this piece of code you think you will have p1 and p3 point to the same object. Yes, they will. But they will have diferent representation objects in p1 and p3. So when p3 is destroyed

the underlying CSomething object will be destroyed too and p1 will point to invalid memory. So I have a simple **tip: If you use SmartPtr never use real pointers to underlying objects** like in the code above. But if you want you can still use real pointers when dealing with other objects.

Let's look at this code:

```
LPSOMETHING     p1 = new CSomething;
SomeFunc( p1 );

void    SomeFunc( CSomething* pSth ) {
        LPSOMETHING     p3 = pSth;
}
```

p3 will create its own representation, ie. its own reference counter and when SomeFunc exits, p3 will free the memory pointed by p1. Instead of this function definition you'd better define it this way.

```
void    SomeFunc( LPSOMETHING pSth );
```

I talk about these to notify you that calling SmartPtr ***constructor*** or ***operator*** = with T* parameter creates new representation object and this will lead to "strange" behaviour of SmartPtr.

The mentioned above weakness of the SmartPtr behaviour could be solved if we add static table of associations - T* <-> CxxxxxRep* and every time SmartPtr gets T* we could search in this table and get appropriate CxxxRep object if it is already created. But this will lead to more memory and CPU overhead. And since now I think it is better to follow mentioned above rule than overheading.

SmartPtr is as efficient as regular pointer and integer values.

Here is a table with some sizeof() results:

	Size in bytes
sizeof(SmartPtr)	4
sizeof(CSomething*)	4
sizeof(int)	4

SmartPtr objects have the same size as pointers and int values. So, passing a SmartPtr objects as function (method) arguments or returning SmartPtr object as function result is as efficient as doing the same with regular pointer or int value.

SmartPtr objects can accept different pointer types.

SmartPtr class has a non template base class SmartPtrBase, which is made argument of various constructors and assignment operators. Thus a code like this is possible to be used

```
class   B {
....
};

class   A : public B {
```

```
....
};

class   C {
....
};

RefCountPtr      a = new A;
RefCountPtr      b = a;
b = a;
```

Even this is very good feature in cases where you need a pointer to base class to hold objects of inherited classes you may have a code like this

```
RefCountPtr      a = new A;
RefCountPtr      b = new B;
SyncPtr c = a;
a = b;
a = c;
c = a;

c = b;
```

And the compiler will not warn you there are implicit type conversions. Note that there is no relation between **class C** and **class B**.

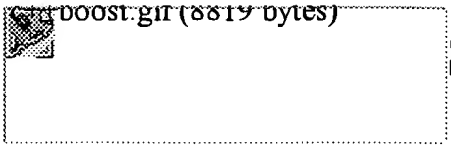
In fact, SmartPtr implicitly passes underlying representation objects in such assignments. The behaviour of the pointer depends on the underlying representation object. So, no matter what is the intention of the holder pointer (in the example above "c" is a synchronization pointer) the pointer behaviour depends on the type of the representation object creator.

In the example above, after the last line (c = b;), "c" will hold the object pointed by "b", and even "c" is declared as "synchronization" pointer when it deals with its object it will behave as "reference counting" pointer, because the original creator is "b" which is of such type.

The enclosed demo project is a simple console application and demonstrates different situations and usage of SmartPtr. It is created by VC ++ 5.0. SmartPtr.h can be compiled with warning level 4.

Downloads

Download demo project - 12 Kb
Download source - 4 Kb



Smart Pointers

Smart pointers are classes which store pointers to dynamically allocated (heap) objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners.

Conceptually, smart pointers are seen as owning the object pointed to, and thus responsible for deletion of the object when it is no longer needed.

The header boost/smart_ptr.hpp provides four smart pointer template classes:

scoped_ptr	Simple sole ownership of single objects.
scoped_array	Simple sole ownership of arrays.
shared_ptr	Object ownership shared among multiple pointers
shared_array	Array ownership shared among multiple pointers.

These classes are designed to complement the C++ Standard Library `auto_ptr` class.

They are examples of the "resource acquisition is initialization" idiom described in Bjarne Stroustrup's "The C++ Programming Language", 3rd edition, Section 14.4, Resource Management.

A test program (smart_ptr_test.cpp) is provided to verify correct operation.

A page on Smart Pointer Timings will be of interest to those curious about performance issues.

Common requirements

These smart pointer classes have a template parameter, `T`, which specifies the type of the object pointed to by the smart pointer. The behavior of all four classes is undefined if the destructor or operator delete for objects of type `T` throws exceptions.

Exception safety

Several functions in these smart pointer classes are specified as having "no effect" or "no effect except such-and-such" if an exception is thrown. This means that when an exception is thrown by an object of one of these classes, the entire program state remains the same as it was prior to the function call which resulted in the exception being thrown. This amounts to a guarantee that there are no detectable side effects. Other functions never throw exceptions. The only exception ever thrown by functions which do throw (assuming `T` meets the Common requirements) is `std::bad_alloc` , and that is thrown only by functions which are explicitly documented as possibly throwing `std::bad_alloc` .

Exception-specifications

Exception-specifications are not used; see exception-specification rationale.

All four classes contain member functions which can never throw exceptions, because they neither throw exceptions themselves nor call other functions which may throw exceptions. These members are indicated by a comment: `// never throws .`

Functions which destroy objects of the pointed to type are prohibited from throwing exceptions by the Common requirements.

History and acknowledgements

November, 1999. Darin Adler provided operator ==, operator !=, and std::swap and std::less specializations for shared types.

September, 1999. Luis Coelho provided shared_ptr::swap and shared_array::swap

May, 1999. In April and May, 1999, Valentin Bonnard and David Abrahams made a number of suggestions resulting in numerous improvements. See the revision history in `smart_ptr.hpp` for the specific changes made as a result of their constructive criticism.

Oct, 1998. In 1994 Greg Colvin proposed to the C++ Standards Committee classes named **auto_ptr** and **counted_ptr** which were very similar to what we now call **scoped_ptr** and **shared_ptr**. The committee document was 94-168/N0555, Exception Safe Smart Pointers. In one of the very few cases where the Library Working Group's recommendations were not followed by the full committee, **counted_ptr** was rejected and surprising transfer-of-ownership semantics were added to **auto-ptr**.

Beman Dawes proposed reviving the original semantics under the names **safe_ptr** and **counted_ptr** at an October, 1998, meeting of Per Andersson, Matt Austern, Greg Colvin, Sean Corfield, Pete Becker, Nico Josuttis, Dietmar Kühl, Nathan Myers, Chichiang Wan and Judy Ward. During the discussion, the four class names were finalized, it was decided that there was no need to exactly follow the **std::auto_ptr** interface, and various function signatures and semantics were finalized.

Over the next three months, several implementations were considered for **shared_ptr**, and discussed on the boost.org mailing list. The implementation questions revolved around the reference count which must be kept, either attached to the pointed to object, or detached elsewhere. Each of those variants have themselves two major variants:

- Direct detached: the shared_ptr contains a pointer to the object, and a pointer to the count.
- Indirect detached: the shared_ptr contains a pointer to a helper object, which in turn contains a pointer to the object and the count.
- Embedded attached: the count is a member of the object pointed to.
- Placement attached: the count is attached via operator new manipulations.

Each implementation technique has advantages and disadvantages. We went so far as to run various timings of the direct and indirect approaches, and found that at least on Intel Pentium chips there was very little measurable difference. Kevlin Henney provided a paper he wrote on "Counted Body Techniques." Dietmar Kühl suggested an elegant partial template specialization technique to allow users

to choose which implementation they preferred, and that was also experimented with.

But Greg Colvin and Jerry Schwarz argued that "parameterization will discourage users", and in the end we choose to supply only the direct implementation.

See the Revision History section of the header for further contributors.

Revised 27 Jul 2000

```
//============================================================================
// Contents:
//
//  Definition of countable_ptr template class and its member functions.
//  countable_ptr implements a reference counted smart pointer for pointers to
//  types that satisfy countable requirements (defined below).
//
//  The code demonstrates a generalised implemention of the Counted Body idiom
//  (aka Attached Counted Handle/Body idiom). The benefit of an approach based
//  on generic programming is that the requirements for counting are separated
//  from the implementation mechanism for counting. This open approach means
//  that countable_ptr is suitable for a wide variety of different
//  implementations, and may adapt to existing ones, rather than requiring a
//  new smart pointer type for each strategy.
//
//  The code is in many ways a traditional implementation of the Counted Body
//  idiom in that it is intrusive, ie countability is acquired explicitly and
//  statically by derivation for a given class. This means that the reference
//  count is explicitly a part of objects even if it is unused (eg in the case
//  of composite members or auto variables), and that only classes over which
//  you have control can use it. However, by satisfying the countability
//  requirements for countable_ptr no new smart pointer class is required to
//  use classes derived from countability.
//
// History:
//
//  Initial version created by Kevlin Henney, kevlin@acm.org, January 1998.
//
// Permissions:
//
//  Copyright Kevlin Henney, 1998. All rights reserved.
//
//  Permission to use, copy, modify, and distribute this software for any
//  purpose is hereby granted without fee, provided that this copyright and
//  permissions notice appear in all copies and derivatives, and that no
//  charge may be made for the software and its documentation except to cover
//  cost of distribution.
//
//  This software is provided "as is" without express or implied warranty.
//
// Notes:
//
//  As there is no requirement that a countable type must be a class, typename
//  is used to introduce type parameters rather than class. If your compiler
//  does not yet support this use of typename, you can replace it with class.
//
//  For practical reasons the global, rather than a named namespace, has been
//  used to hold the contents of this header. In future the use of a separate
//  namespace is preferred and recommended.
//
//  This code has been written to conform to standard C++ (in final draft
//  status at the time of writing). It has been compiled successfully using
//  the operational subset of standard features implemented by Microsoft
//  Visual C++ 5.0.
//============================================================================

#ifndef COUNTABLE_PTR_INCLUDED
#define COUNTABLE_PTR_INCLUDED

//----------------------------------------------------------------------------
// Description:
//
//  Definition of countable_ptr template class, which implements a reference
//  counted smart pointer template for types that satisfy countable
//  requirements. The class is a concrete class not intended as a base class.
```

```
//
// Requirements:
//
//  For a type to be countable it must satisfy the following requirements,
//  where ptr is a non-null pointer to a single object (ie not an array) of
//  the type, and #function indicates number of calls to function(ptr):
//
//    +==================+====================+============================+
//    | expression       | return type        | semantics and notes        |
//    +==================+====================+============================+
//    | aquire(ptr)      | no requirement     | post: acquired(ptr)        |
//    +------------------+--------------------+----------------------------+
//    | release(ptr)     | no requirement     | pre:  acquired(ptr)        |
//    |                  |                    | post: acquired(ptr) ==     |
//    |                  |                    |   #acquire > #release      |
//    +------------------+--------------------+----------------------------+
//    | acquired(ptr)    | convertible to bool | return: #acquire > #release |
//    +------------------+--------------------+----------------------------+
//    | dispose(ptr, ptr) | no requirement    | pre:  !acquired(ptr)       |
//    |                  |                    | post: *ptr no longer usable |
//    +==================+====================+============================+
//
//  Note that the two arguments to dispose are to support selection of the
//  appropriate type safe version of the function to be called. In the general
//  case the intent is that the first argument determines the type to be
//  deleted, and would typically be templated, while the second selects which
//  template to use, eg by conforming to a specific base class.
//
//  In addition the following requirements must also be satisfied, where null
//  is a null pointer to the countable type:
//
//    +====================+====================+====================+
//    | expression         | return type        | semantics and notes |
//    +====================+====================+====================+
//    | aquire(null)       | no requirement     | action: none       |
//    +--------------------+--------------------+--------------------+
//    | release(null)      | no requirement     | action: none       |
//    +--------------------+--------------------+--------------------+
//    | acquired(null)     | convertible to bool | return: false     |
//    +--------------------+--------------------+--------------------+
//    | dispose(null, null) | no requirement    | action: none       |
//    +====================+====================+====================+
//
//  Note that there are no requirements on these functions in terms of
//  exceptions thrown or not thrown, except that if exceptions are thrown the
//  functions themselves should be exception safe.
//
//  In principle these functions may be used on an object independently of
//  countable_ptr, but for a number of reasons this is inadvisable and no
//  guarantees (except "you'll be sorry") are made for mixed use (eg using
//  both countable_ptr and manual calls of countable functions on an object).
//
// Notes:
//
//  The constructor taking a single pointer has been made explicit to prevent
//  accidental conversions, but the explicit keyword can be removed if your
//  compiler does not support it without affecting the intent.
//
//  For brevity, equality and Boolean operators have been omitted, but are
//  simple to add according to the model you wish to support.
//
//  For brevity and portability, member templates for the copy constructor
//  and for the copy assignment operator have been omitted. These are simple
//  to implement, and are identical in behaviour to the regular copy
//  constructor and copy assignment operator [NB: use the get member function
```

```
//  to gain access to the held pointer].
//
//  Related member functions (method categories) are placed under their own
//  commented access specifiers.
//
//  The naming convention adopted is in part based on the standard library,
//  ie get, clear and assign take their names and basic behaviour from the
//  spec for auto_ptr and basic_string.
//------------------------------------------------------------------------------

template<typename countable_type>
class countable_ptr
{
public: // construction and destruction

    explicit countable_ptr(countable_type *);
    countable_ptr(const countable_ptr &);
    ~countable_ptr();

public: // access

    countable_type *operator->() const throw();
    countable_type &operator*() const throw();
    countable_type *get() const throw();

public: // modification

    countable_ptr &clear();
    countable_ptr &assign(countable_type *);
    countable_ptr &assign(const countable_ptr &);
    countable_ptr &operator=(const countable_ptr &);

private: // representation

    countable_type *body;

};

//------------------------------------------------------------------------------
// Description:
//
//  Definition of countable_ptr template class member functions.
//
// Notes:
//
//  Definitions could be placed in a separate header file or, for systems that
//  support separate compilation of templates, in a source file.
//
//  A number of functions are likely candidates for explicit inlining, but for
//  the purposes of demonstration such an optimisation has not been deemed
//  necessary.
//------------------------------------------------------------------------------

template<typename countable_type>
countable_ptr<countable_type>::countable_ptr(countable_type *initial)
  : body(initial)
{
    acquire(body);
}

template<typename countable_type>
countable_ptr<countable_type>::countable_ptr(const countable_ptr &other)
  : body(other.body)
{
    acquire(body);
```

```
}

template<typename countable_type>
countable_ptr<countable_type>::~countable_ptr()
{
    clear();
}

template<typename countable_type>
countable_type *countable_ptr<countable_type>::operator->() const throw()
{
    return body;
}

template<typename countable_type>
countable_type &countable_ptr<countable_type>::operator*() const throw()
{
    return *body;
}

template<typename countable_type>
countable_type *countable_ptr<countable_type>::get() const throw()
{
    return body;
}

template<typename countable_type>
countable_ptr<countable_type> &countable_ptr<countable_type>::clear()
{
    return assign(0);
}

template<typename countable_type>
countable_ptr<countable_type> &
countable_ptr<countable_type>::assign(countable_type *rhs)
{
    // set to rhs (note that this sequence is self assignment safe)
    acquire(rhs);
    countable_type *old_body = body;
    body = rhs;

    // tidy up
    release(old_body);
    if(!acquired(old_body))
    {
        dispose(old_body, old_body);
    }

    return *this;
}

template<typename countable_type>
countable_ptr<countable_type> &
countable_ptr<countable_type>::assign(const countable_ptr &rhs)
{
    return assign(rhs.body);
}

template<typename countable_type>
countable_ptr<countable_type> &
countable_ptr<countable_type>::operator=(const countable_ptr &rhs)
{
    return assign(rhs);
}
```

```
#endif
```

auto_ptr

ObjectSpace

A templatized smart pointer class.

The `auto_ptr` class is used to automatically manage pointers to dynamically created objects. When an `auto_ptr` is destroyed, the object it points to is also destroyed. Only one `auto_ptr` can own an object at a time; copying an `auto_ptr` will transfer the object pointer and ownership of the object to the destination `auto_ptr` . Therefore, `auto_ptr` s can be used safely as return values for functions.

Library

Standards<ToolKit>

Declaration

```
#include <memory>
```

template< class T >
class auto_ptr

Interface

Constructor
explicit auto_ptr(T* *ptr*)
Constructs an auto pointer to manage the object pointed to by *ptr* (default `0`).

Constructor
auto_ptr(auto_ptr< T >& *other*)
Constructs an auto pointer and assumes ownership of the object managed by *other* .

Destructor
~auto_ptr()
Destroys the auto pointer and the object it manages.

=
auto_ptr< T >& operator=(auto_ptr< T > *other*)
Assumes ownership of the object managed by *other* . If the auto pointer already manages an object, destroys the original object first.

*
T& operator*() const
Returns a reference to the auto pointer's managed object.

->
T* operator->() const
Returns a pointer to the object auto pointer manages.

get
T* get() const
Returns a pointer to the object auto pointer manages.

release
T* release()

Returns a pointer to the auto pointer's managed object and releases ownership.

Standard C++ Library Module Reference Guide
Rogue Wave web site: Home Page | Main Documentation Page

auto_ptr

Module: Standard C++ Library **Library:** General utilities

Does not inherit

Local Index

Members

auto_ptr()	get()	operator*()	release()
auto_ptr_ref	operator auto_ptr<Y>()	operator->()	reset()
element_type	operator auto_ptr_ref<Y>()	operator=()	~auto_ptr()

Summary

A simple smart pointer class

Synopsis

```
#include <memory>

namespace std {
  template <class X> class auto_ptr;
}
```

Description

The class template specialization ***auto_ptr*** holds onto a pointer obtained via `new()` and then deletes that object when the ***auto_ptr*** object itself is destroyed. ***auto_ptr*** can be used to make calls to operator `new()` exception-safe.

The ***auto_ptr*** class has semantics of strict ownership: an object may be safely pointed to by only one ***auto_ptr***, so copying an ***auto_ptr*** copies the pointer and transfers ownership to the destination if the source had already had ownership.

Interface

```
namespace std {

  template <class Y> struct auto_ptr_ref {};
  template <class X> class auto_ptr {
     public:
       typedef X element_type;

       // construct/copy/destroy
       explicit auto_ptr (X* p = 0) throw();
       auto_ptr(auto_ptr<X>&) throw ();
       template <class Y>
       auto_ptr(auto_ptr<Y>&) throw();
       auto_ptr<X>& operator=(auto_ptr<X>&) throw();
       template <class Y>
       auto_ptr<X>& operator= (auto_ptr<Y>&) throw();
       ~auto_ptr() throw();

       // members
       X& operator* () const throw();
       X* operator-> () const throw();
       X* get () const throw();
       X* release() throw();
       void reset(X* p = 0) throw();

       // conversions
       auto_ptr(auto_ptr_ref<X>) throw();
       template <class Y> operator auto_ptr_ref<Y>() throw();
       template <class Y> operator auto_ptr<Y>() throw();
  };
}
```

Struct auto_ptr_ref

```
template <class Y>
struct auto_ptr_ref ;
```

A namespace-scope struct template that holds a reference to an ***auto_ptr***. An ***auto_ptr_ref*** can only be constructed within an ***auto_ptr*** using a reference to an ***auto_ptr***. It prevents unsafe copying.

Typedef

```
typedef X element_type ;
```

The type of element pointed to by ***auto_ptr***.

Constructors

```
explicit
auto_ptr (X* p = 0) throw();
```

Constructs an object of class ***auto_ptr<X>***, initializing the held pointer to `p`, and acquiring ownership of that

pointer. p must point to an object of class X, a class derived from X for which delete p is defined and accessible, or p must be a null pointer.

```
auto_ptr (auto_ptr<X>& a) throw();
template <class Y>
auto_ptr (auto_ptr<Y>& a) throw();
```

Constructs an object of class ***auto_ptr<X>***, and copies the argument a to *this. If a owned the underlying pointer, then *this becomes the new owner of that pointer.

For the constructor template, each specialization requires that a pointer to Y be implicitly convertible to pointer to element_type.

```
auto_ptr (auto_ptr_ref<X> r) throw();
```

Constructs an ***auto_ptr*** from an ***auto_ptr_ref***.

Destructors

```
~auto_ptr () throw();
```

Deletes the underlying pointer.

Operators

```
auto_ptr<X>& operator= (auto_ptr<X>& a) throw();
template <class Y>
auto_ptr<X>& operator= (auto_ptr<Y>& a) throw();
```

Copies the argument a to *this. If a owned the underlying pointer, then *this becomes the new owner of that pointer. If *this already owned a pointer, that pointer is deleted first. The argument a is reset to zero.

For the function template, each specialization requires that a pointer to Y be implicitly convertible to pointer to element_type.

```
X&
operator* () const throw();
```

Returns a reference to the object to which the underlying pointer points.

```
X*
operator-> () const throw();
```

Returns the underlying pointer.

```
template <class Y>
operator auto_ptr_ref<Y> () throw();
```

Constructs an ***auto_ptr_ref*** from *this and returns it.

```
template <class Y>
operator auto_ptr<Y> () throw();
```

Constructs a new ***auto_ptr*** using the underlying pointer held by *this. Calls release() on *this, so *this no longer possesses the pointer. Returns the new ***auto_ptr***.

Member Functions

```
X*
get() const throw();
```

Returns the underlying pointer.

```
X*
release() throw();
```

Releases ownership of the underlying pointer and returns that pointer. The `*this` object is left holding a null pointer.

```
void
reset(X* p = 0) throw();
```

Sets the underlying pointer to `p`. If non-null, deletes the old underlying pointer.

Example

```
//
// auto_ptr.cpp
//

#include <iostream>   // for cout, endl
#include <memory>     // for auto_ptr


// A simple structure.
class X
{
    int i_;

public:

    X (int i) : i_ (i) {
        std::cout << "X::X (" << i_ << ')' << std::endl;
    }

    ~X () {
        std::cout << "X::~X [" << i_ << ']' << std::endl;
    }

    int get () const { return i_; }
};


int main ()
{
    // a implicitly initialized to 0 (the null pointer)
    std::auto_ptr<X> a;

    // Establish a scope.
    if (1) {
        // b will hold a pointer to an X.
        std::auto_ptr<X> b (new X (12345));

        // a will now be the owner of
        // the underlying pointer.
        a = b;
    }
```

```
    std::cout << "b destroyed" << std::endl;

        // Output the value contained by
        // the underlying pointer.
#ifndef _RWSTD_NO_NONCLASS_ARROW_RETURN
        std::cout << a->get () << std::endl;
#else
        std::cout << (*a).get () << std::endl;
#endif

    // The pointer will be deleted when a is destroyed
    // on leaving scope.
    return 0;
}

Program Output:
```

X::X (12345)

b destroyed

12345

X::~X [12345]

Standards Conformance

ISO/IEC 14882:1998 -- International Standard for Information Systems -- Programming Language C++, Section 20.4.5

◄ Top Contents Index ►